>>> Julia para Pythonistas

Name: Melissa Weber Mendonça

Date: PythonSul 2018

>>> Quem é Julia?

```
[melissa@oneesk ~]$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: https://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.6.2 (2017-12-13 18:08 UTC)
 _/ |\__'_|_|_|\__'_|  |
|__/                   |  x86_64-pc-linux-gnu

julia>
```

* Julia é uma linguagem dinâmica, de alto nível e alto desempenho (disponibilizada em 2012)
* Ideal para computação científica e numérica; também ciência de dados
* Bibliotecas adicionais podem ser instaladas via gerenciador de pacotes: `Pkg.add("pacote")`

>>> Características da linguagem

* Free e open source (licença MIT)
* A biblioteca padrão é escrita em Julia
* Multiparadigma (quase funcional; baseada em Scheme, MATLAB)
* Tem um compilador sofisticado (JIT/LLVM)
* Suporta a computação paralela e distribuída
* Permite precisão numérica arbitrária
* *Multiple Dispatch*: tipos são a chave

>>> Uma linguagem dinâmica, mas...

* Código não-vetorizado é rápido
* Sistema de tipagem elegante e extensível, com conversão e promoção (o que possibilita o multiple dispatch)
* Tipos definidos pelo usuário são tão eficientes quanto os tipos built-in
* Possibilidade de chamar funções em C diretamente (sem wrappers ou API's)
* Possibilidade de gerenciar outros processos
* Macros estilo Lisp e outras possibilidades em metaprogramação
* *Lightweight "green" threading (coroutines)*

https://julialang.org/benchmarks/

>>> A hierarquia de tipos

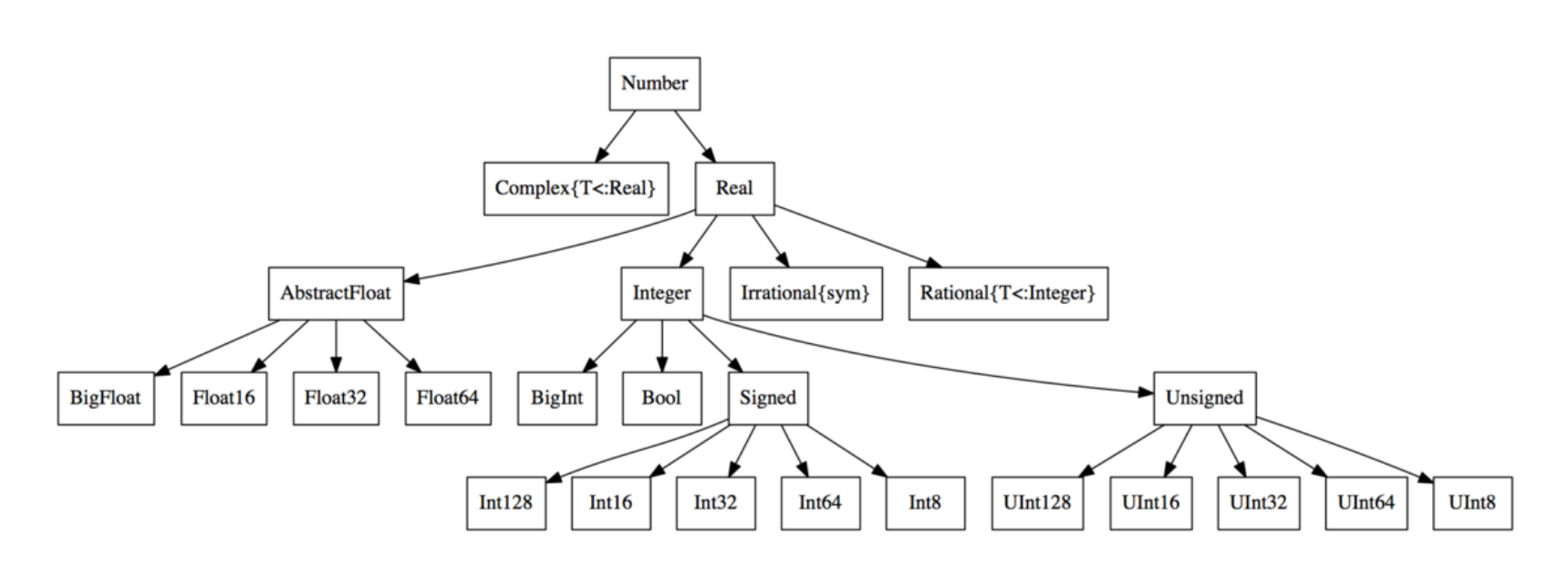

Figura: Hierarquia de tipos numéricos

Tipos abstratos servem para definir funções, mas só os tipos concretos (folhas da árvore) podem conter valores.

>>> "Code Selection" vs. OOP[1]

O método é atrelado à função: não ao argumento.

Exemplo: Ao somar duas matrizes A e B, podemos pensar em Python em algo do tipo

A.plus(B) ← *single dispatch!*

Em Julia, temos

plus(A,B)

A grande diferença é que estamos considerando o tipo de TODOS os argumentos ao escolher a função plus correta para ser utilizada com o tipo de argumentos que temos.

[1] Veja [Bezanson et al., 2017]

>>> Multiple Dispatch

Os métodos são definidos de maneira (possivelmente) diferente para cada combinação de argumentos de entrada.

Exemplo:

```
collide_with(x::Asteroid, y::Asteroid) = ...
collide_with(x::Asteroid, y::Spaceship) = ...
collide_with(x::Spaceship, y::Asteroid) = ...
collide_with(x::Spaceship, y::Spaceship) = ...
```

>>> Coleções e "A Vingança dos Loops"

* Julia tem arrays, tuplas, dicionários
* "Ah mas os índices começam em 1..."

```
x = OffsetArray(Float64, 0:9)
```

* E a vetorização? Não é necessária pra melhorar o desempenho se já sabemos o tipo de variável que está sendo usada![2]

Pode usar for sem medo de ser feliz!

---

[2] "Users of traditional high-level computing languages know that vectorization improves performance. Do most users know exactly why vectorization is so useful? It is precisely because, by vectorizing, the user has promised the computer that the type of an entire vector of data matches the very first element." [Bezanson et al., 2017]

>>> Metaprogramação

Julia tem um sistema de macros, símbolos e *expressions*, que permitem escrever código julia em julia.

Exemplo:[3]

```
julia> foo = "hello"
julia> :foo
:foo
julia> typeof(:foo)
Symbol
julia> typeof(foo)
String
julia> eval(:foo)
"hello"
julia> eval(foo)
"hello"
julia> eval("foo")
"foo"
```

[3] https://stackoverflow.com/questions/23480722/what-is-a-symbol-in-julia

>>> Macros de tradução

Permitem ao usuário observar a tradução de um pedaço de código em julia para níveis mais baixos[4]:

* @code_lowered
* @code_typed
* @code_llvm
* @code_native

[4] https://goo.gl/FUko5L(StackOverflow)

>>> Não precisa abandonar o Python!

Com PyCall.jl, é possível usar Python diretamente dentro do julia:

```
julia> @pyimport scipy.optimize as so
julia> so.newton(x -> cos(x) - x, 1)
```

>>> Por que escolher Julia para Data Science?[5]

Why Does Julia Work So Well?

There is an obvious reason to choose Julia:

> it's faster than other scripting languages, allowing you to have the rapid development of Python/MATLAB/R while producing code that is as fast as C/Fortran

Newcomers to Julia might be a little wary of that statement.

1. Why not just make other scripting languages faster? If Julia can do it, why can't others?
2. How do you interpert Julia benchmarks to confirm this? (This is surprisingly difficult for many!)
3. That sounds like it violates the No-Free-Lunch heuristic. Is there really nothing lost?

Many people believe Julia is fast **because it is Just-In-Time (JIT) compiled** (i.e. every statement is run using compiled functions which are either compiled right before they are used, or cached compilations from before). This leads to questions about what Julia gives over JIT'd implementations of Python/R (and MATLAB by default uses a JIT). These JIT compilers have been optimized for far longer than Julia, so why should we be crazy and believe that somehow Julia quickly out-optimized all of them? However, that is a complete misunderstanding of Julia. What I want show, in a very visual way, is that Julia is fast because of its design decisions. The core design decision, **type-stability through specialization via multiple-dispatch** is what allows Julia to be very easy for a compiler to make into efficient code, but also allow the code to be very concise and "look like a scripting language". This will lead to some very clear performance gains.

But what we will see in this example is that Julia does not always act like other scripting languages. There are some "lunches lost" that we will have to understand. Understanding how this design decision effects the way you must code is crucial to producing efficient Julia code.

To see the difference, we only need to go as far as basic math.

[5] http://ucidatascienceinitiative.github.io/IntroToJulia/Html/WhyJulia

>>> Tem muito mais pra dizer...

* Julia é jovem! Muda muito rápido
* Documentação ainda está incompleta
* Pacotes estão surgindo a todo momento
* Só vem ♡

>>> Tem muito mais pra dizer...

* Julia é jovem! Muda muito rápido
* Documentação ainda está incompleta
* Pacotes estão surgindo a todo momento
* Só vem ♡[6]

[6] (Fala sério, dá pra chamar uma variável de ♡!!!!)

telegram: melissawm
twitter: @melissawm
email: melissa.mendonca@ufsc.br

>>> Bibliografia I

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017).
Julia: A fresh approach to numerical computing.
SIAM Review, pages 65--98.